Lightweight Resource Exposure Framework for Efficient Service and Resource Orchestration in the Cloud-Edge Continuum

Abd Elghani Meliani
communication systems
Eurecom
Biot, France
abd-elghani.meliani@eurecom.fr

Adlen Ksentini
communication systems
Eurecom
Biot, France
adlen.ksentini@eurecom.fr

***Abstract*—The Cloud Edge Continuum (CEC) opens up new opportunities for deploying applications across a wide array of resources, ranging from centralized cloud infrastructures to edge and far-edge nodes. However, orchestrating resources in this heterogeneous and dynamic environment requires innovative approaches beyond traditional cloud or network function virtualization (NFV) systems. In this paper, we introduce a novel orchestration framework for the CEC that separates service orchestration from resource orchestration. This separation is crucial for managing the ever-changing and diverse nature of CEC resources, particularly at the edge and far-edge levels. To facilitate this, it is essential for the orchestration and management plane of the CEC to incorporate a Resource Exposer (RE) component, which is the primary contribution of this work. Additionally, a Central Resource Discovery (RD) Module aggregates data from multiple REs, providing a comprehensive global view of the registered infrastructures and available resources. All components of the framework run in containers, ensuring compatibility with containerized environments and seamless integration across Kubernetes, openshift, KubeEdge, and K3S platforms. The experimentation results demonstrate that the proposed system is highly efficient, with minimal CPU and memory consumption, even when deployed on low-resource devices like edge nodes. The results show that the RE achieves low-latency responses and scales well under high-frequency data collection, making it a viable solution for orchestrating resources in the cloud-edge continuum.**

***Index Terms*—Cloud Edge Continuum, Resource Orchestration, Resource Exposer**

I. Introduction

CEC is the next trend in cloud computing. It involves the strategic deployment of applications and services across a continuum of resources, which includes centralized cloud, edge cloud, and far-edge (or extreme edge) nodes. By enabling CEC, we can unlock the potential of applications requiring low latency (which operate at far-edge or edge nodes) or those that need to process data locally (such as Federated Learning, video analysis, etc.). CEC is also a key enabler for running cloud-native applications that typically rely on microservice architectures. Some microservices run as close as possible to the end-user or the data source, while others, such as the front-end interfaces, may run in the centralized cloud. Meanwhile, the management and orchestration functions of CEC resources differ from the classical approaches used in Cloud Computing or NFV [1]. The latter relies on a centralized orchestrator, whereas in CEC, orchestration must handle a distributed system composed of various computing nodes, each with different characteristics. The centralized cloud has abundant resources but high round-trip time (RTT) access (around 200 ms round trip), edge computing offers fewer resources but lower RTT access (around 50 ms), and far-edge nodes have minimal computing resources but practically zero latency. Additionally, far-edge resources are volatile in terms of availability, as they can be mobile (e.g., Unmanned Aerial Vehicles - UAVs or drones) or have battery constraints (e.g., IoT devices). To overcome the challenges posed by the orchestration of CEC resources, we propose a new framework that adopts the approach introduced in [2], which involves separating service orchestration from resource orchestration. At the service (or application) level, the orchestration system ensures the application's lifecycle management (LCM) and adherence to the service level agreement (SLA); it interacts with the resources not directly, but through a Resource Orchestrator (RO) that handles and manages the resources. This separation of concerns allows for an abstraction that enables the service orchestrator (SO) to manage services without directly handling resources. The resources can be dynamically discovered and managed by the RO. Thus, volatile resources can appear and disappear transparently to the SO, which only sees the available resources exposed by the RO. A key feature ensuring the separation of service and resource orchestration is the resource exposure mechanism. This allows the RO to track the resource pools it can use. Resource exposure involves a continual flow of information provided by the local resource manager (such as Kubernetes or an SDN controller) indicating the amount, type, and availability of resources. In the case of volatile far-edge nodes, whenever they are available, the local resource

manager pushes this information to the RO. Resource exposure is critical for achieving efficient CEC resource orchestration, given the heterogeneity of computing resources that form the computing continuum. Based on resource exposure, the RO abstracts resource usage for the SO, which views resources as a pool sorted by geographical location, type, latency capabilities, etc. In this paper, we propose a lightweight resource exposure framework for CEC that can be integrated with various Local Management Systems (LMS) (such as Kubernetes, K3S, and Kubedge). This framework is highly relevant for CEC and ensures the separation of resource and service orchestration paradigms. The contributions of this paper are:

1) *A novel CEC orchestration paradigm:* The new framework separates between service and resource orchestration to deal with the volatility of computing nodes (like far-edge nodes) and the heterogeneous resources of the CEC.
2) *Resource Exposer:* Built on a plugin-based architecture to ensure technological agnosticism, this component is deployed within each infrastructure to expose local resource utilization data.
3) *Central Resource Discovery Module:* This module aggregates data from multiple REs, offering various entities a comprehensive global view of the current state of all the registered infrastructures.

The rest of this paper is structured as follows: Section II reviews related work, while Section III outlines our key contributions. Section IV presents the performance evaluation and the results of our proposed solution. Finally, Section V concludes the paper and discusses future directions.

II. Related Work

Numerous approaches have been proposed for systems monitoring in distributed environments. One prominent example is the DECOR system [3], which utilizes a distributed method for resource monitoring, particularly in network-based applications like redundancy elimination and traffic sampling. By distributing monitoring tasks across various network nodes, DECOR avoids the bottlenecks typically associated with centralized controllers. Similarly, Dprof [4] presents a lightweight, distributed profiling system designed to trace Remote Procedural Calls (RPC) operations and identify performance bottlenecks in complex distributed systems. Dprof gathers and analyzes RPC traces from heterogeneous components, storing them in a distributed data store for further analysis. While effective for debugging performance issues at the RPC level in systems, Dprof focuses on system-level event profiling and lacks comprehensive infrastructure monitoring capabilities. A scalable monitoring framework introduced in [5] specifically tackles the challenges of monitoring 5G network slices, with emphasis on resource isolation, multi-tenancy, and the integration of different technological domains such as RAN and cloud. This framework employs slice-specific collectors that follow the lifecycle of each network slice, ensuring efficient data collection across diverse domains.

Recently, both industry and academia have been advocating for standardized RE solutions to improve infrastructure-aware service deployment, as emphasized by a recent proposal from the Internet Engineering Task Force (IETF) [6]. While we share the broader goal of improving resource management, service discovery, and system bottleneck detection with existing solutions, we believe we are the first to introduce a unified RE tailored for heterogeneous systems.

III. Service and resource orchestration separation in CEC: Resource exposer approach

A. CEC orchestration

As mentioned earlier, this work presents a novel resource exposure approach to simplify the orchestration and management of CEC's heterogeneous resources. A high-level view of the proposed CEC orchestration and management framework, which distinguishes service management from resource management, is illustrated in Figure 1.

In the figure, we illustrate three key components of the CEC orchestration and management system. The first component is the service (or application) orchestrator, which manages the lifecycle of applications and their microservices. This includes deployment (placement), runtime (ensuring that the Service Level Agreement (SLA) is maintained by executing the decision-making algorithms [7], such as migration or scalable requests), and deletion. The SO interacts with the RO to get a global view on the infrastructure, and this is to implement LCM decisions through a unified API provided by the latter. In the proposed CEC orchestration system, a key innovation is the abstraction of CEC resources for the SO to accommodate the flexibility and heterogeneity of the CEC infrastructure. This is especially critical given the volatility of far-edge resources and the ability to add computing resources dynamically. This abstraction is achieved by the second key element depicted in the figure, the RO. The RO connects to the computing resources through the LMSs , which manage local resources (deployment of containers, pods, and handling of node resources) and provide an API for Create, Read, Update and Delete (CRUD) operations. The LMSs are discovered by the RO dynamically. Whenever an LMS appears in the CEC, it registers itself with the RO via the exposer, indicating its northbound API. The abstraction achieved by the RO primarily involves: (1) translating the SO Life Cycle Management (LCM) requests, which can be articulated using intents, into an LMS-specific API; and (2) exposing the available computing resources at the CEC so that the SO can make informed LCM decisions.

In this paper, our focus is on the second function. As shown in the figure, the RO periodically gathers the resources available at each LMS using the resource exposure mechanism (detailed in the next section). This information is then sorted and presented to the SO by location and type. The detailed

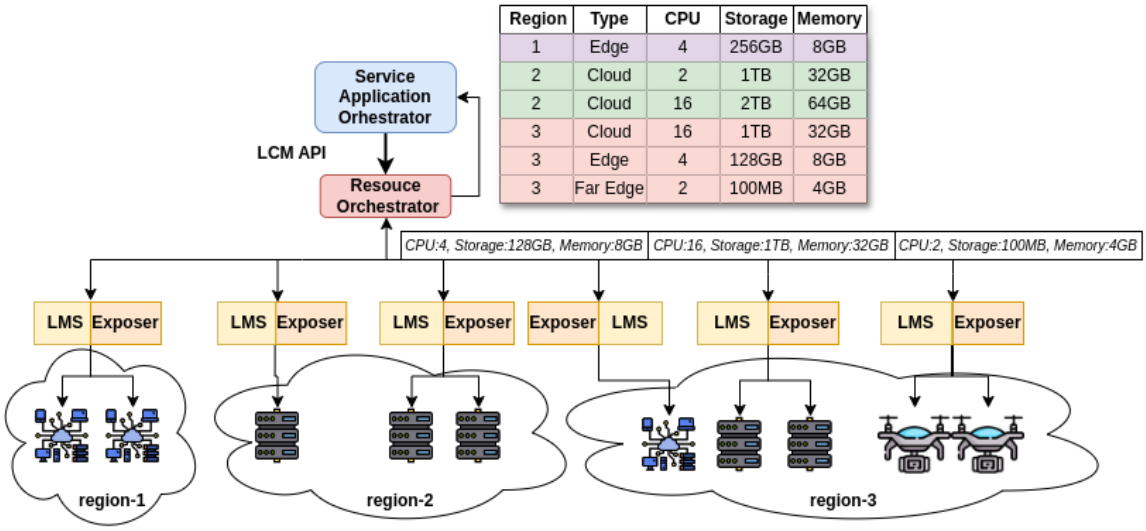

Fig. 1. The higher level architecture of the proposed monitoring framework

information about the available resources of infrastructure nodes is accessible at the resource exposer and orchestrator levels. However, this information is abstracted from the service orchestrator componen. For example,the RO collects resource availability from three LMSs in Region 3 and conveys this information to the SO without detailing individual nodes. It also aggregates resources by type, such as in Region 1. The SO may request to deploy a workload in a specific region by selecting the type of resources (cloud, edge, or far edge) and specifying the required SLA. In the next section, we will present our contribution to developing the resource exposure mechanisms that are essential to ensure the separation of service and resource orchestration in the CEC.

B. Resource exposure

Figure 2 shows the concept of the proposed resource exposure framework. It highlights the main component, the RE, which is deployed on each cluster of computing nodes. The RE is responsible for collecting and providing data on local resource usage and availability. It exposes an API to share the required data with the higher control levels. A gRPC plugin-based system was chosen for the exposer to handle the diversity of source metrics and monitoring solutions, making the system agnostic and compatible with various infrastructures. The plugins serve as intermediaries between the data collector and the existing monitoring solutions. The collected data is temporarily stored in a message broker. This design addresses two main challenges: (1) the RE may need to handle requests from multiple entities simultaneously, especially when the associated computing node is a part of multiple local systems, so quick access to data under peak conditions is essential, and (2) the data only reflects current or near-past resource consumption,

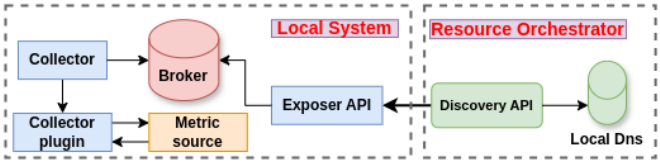

Fig. 2. The lower level architecture of the proposed monitoring framework

eliminating the need for long-term storage. This short-term storage ensures that the Discovery Module can access up-to-date metrics. The system supports multiple brokers, such as RabbitMQ, Kafka, or Redis, allowing the RE to integrate with any existing infrastructure.

To ensure consistency and ease of integration across different computing nodes, we propose a unified data format to represent the metrics. This format ensures that the RE presents metrics in a structured manner, regardless of the infrastructure specifics, providing a consistent view across multiple sources and simplifying data processing and analysis. The exposer collects two main types of metrics: compute and network. For each type, an overview is first provided, detailing the overall availability and consumption of resources across the cluster. This is followed by a more granular breakdown of resource usage in different parts of the system. Figures 3 and 4 illustrate the metrics formats for compute and network resources, respectively.

The exposer tracks key compute metrics, including memory, CPU, and storage. Memory metrics cover total, available, and used memory at cluster and machine levels. CPU metrics include core count, per-core usage, average machine usage, and free capacity. Storage metrics detail used, available, and total storage, I/O operations, and read/write bandwidth over time.

```
"2024-07-19 16:26:22.980972": {
        "numberOfNodes": 7,
         "memory": {
            "provisioned_memory": {
                "totalProvisionedMemory": "value",
                "detailedValues": {
                   "machine01": "value"}},
            "memory_available": {
                "totalAvailableMemory": "value",
                "detailedValues": {
                    "machine01":"value"}},
            "memory_usage": {
                "totalUsedMemory": "value",
                "detailedValues": {
                   "machine01": "value"}}},
        "cpu": {
            "totalProvisionedCpuCores": {
                "totalCpuCores": "value",
                "numberOfCpuCoresPerNode": {
                   "machine01": "value"}},
            "usedCpu": {
                "detailedCpuUsageValues": {
                   "machine01-cpu01":"value"},
                "averageCpuUsageValuesPerNode": {
                  "machine01":"value"}},
            "freeCpu": {
                "detailedFreeCpuValues": {
                  "machine01-cpu01":"value"},
                "averageFreeCpuValuesPerNode": {
                  "machine01":"value"}}},
        "disk": {
            "freeDisk": {
                "totalAvailableDsikSpace": "value",
                "detailedValuesPerNode": {
                "machine01":"value"}},
            "usedDisk": {
                "totalAvailableDsikSpace": "value",
                "detailedValuesPerNode": {
                 "machine01":"value"}},
            "diskThroughput": {
                "read": {
                    "detailedValuesPerStorageUnit": {
                        "machine01-sda": "value",}},
                "write": {
                    "detailedValuesPerStorageUnit": {
                    "machine01-sda":"value"}}}}
}
```


Fig. 3. Example of JSON format for compute resource metrics, including memory, CPU, and storage utilization.


```
"2024-07-19 16:26:22.980972": {
   "clusterNetworkDevicesNumber": "value",
   "packet-loss": {
   "errReceivedPercentage": {
      "machine01:-network-device-bond0": "value",},
      "DropedRecievedPacketsPourcentage": {
          "machine01:-network-device-bond0":
"value",}},
   "latency": {
     "8.8.8.8": {
       "status": "success",
         "data": {
           "Minimum RTT": "value",
           "Average RTT": "value",
           "Maximum RTT": "value",
           "MDEV RTT": "value"}}},
    "jitter": {
      "8.8.8.8": {
       "jitter": "value"}},
    "throughputs": {
      "transmitted_throughput": {
        "machine01:-network-device-bond0": "value"},
      "received_throughput": {
      "machine01:-network-device-bond0": "value"}}
}
```


Fig. 4. Example of JSON format for network resource metrics, including throughput, packet loss, latency, and jitter.

For network metrics, the exposer monitors device throughput (received and transmitted) and packet loss over specified intervals. Additionally, by default, the exposers report latency and jitter between the infrastructure and the RD Module, helping assess network stability. The exposer can also be configured to measure latency between its infrastructure and others, allowing for cross-infrastructure network performance analysis.

In addition, the exposer provides information on the type of energy used and other general infrastructure details, such as whether it is cloud-based, edge, or on-premises. To gain a deeper understanding of how our solution operates, Figure 5 illustrates the interaction workflow between its various components. The figure highlights four key steps:

1) **Exposer Registration:** When a new RE is deployed at LMS, it registers itself with the discovery API, allowing it to easily locate and communicate with it. This self-registration process improves scalability by eliminating the need for the Discovery Module (located at the RO) to actively search for new infrastructures. Instead, each RE automatically handles its own registration, simplifying the process of adding or removing infrastructures. This decentralized method reduces the workload on the Discovery Module and allows the system to adapt flexibly to changes, enabling new computing nodes or clusters to join or leave the network without requiring complex reconfiguration. A Local DNS system is used to manage communication between the discovery module and the

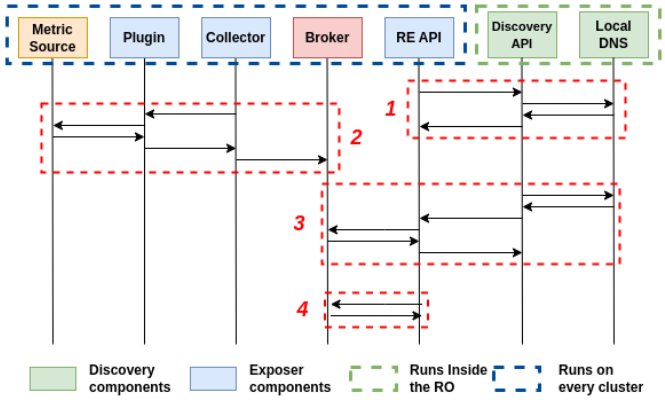

Fig. 5. Interaction Workflow Between The Different Framework Components

exposers by storing the IP addresses and domain names of all RE instances.

2) **Data Collection:** As described earlier, the Collector periodically accesses source metrics via plugins, gathering the necessary data and storing it in the message broker. This process runs in parallel with the RE registration.
3) **Data Exposure:** The Discovery module requests specific information from all available Exposer instances about the infrastructures. The Exposers retrieve the latest data from the brokers and expose it to the Discovery module.
4) **Broker purge:** If an Exposer API goes unused for a certain period, it purges the broker to prevent unnecessary data accumulation. This ensures efficient resource usage and improves overall framework performance by preventing excessive memory consumption.

IV. Evaluation & Results

To assess the performance of our solution, we chose to test it on low-resource devices. This is motivated by the necessity for the RE to operate on every computing node within the continuum. Therefore, testing on low-resource devices, which can represent edge and far-edge nodes, is crucial to ensure that the exposer can function on every node within the continuum. Specifically, we used a Raspberry Pi 4 model, which is equipped with a Broadcom BCM2711 quad-core Cortex-A72 (ARM v8) 64-bit processor running at 1.5 GHz, along with 8 GB of DDR4 RAM. We utilized the 64-bit desktop version of Ubuntu 22.04 as operating system. For the software components, we employed Redis Stack as the broker for the exposer. Additionally, we tested the solution using both K3s v1.31.0 and KubeEdge v1.18.1 as they are generally used for edge devices and both were running on top of containerd v1.7.22. Regarding the tests, we concentrated primarily on the exposer component, as it is the most performance-sensitive part of the system. This component requires minimal resource usage and low latency for serving requests. Over a one-week period, we measured the computing resources utilized by the exposer while varying the data collection interval from 20 seconds to 1 second, which represents a highly computational scenario. Figure 6 (left), which illustrates CPU consumption, shows that as the data collection interval decreases to 1 second, both the exposer and the broker consume less than 0.04 CPU. This indicates that even in high-frequency data collection scenarios, both components maintain relatively low CPU usage, making them compatible with low-resource devices like edge devices. Figure 6 (right), which reflects memory usage (in bytes), reveals that the exposer experiences minimal fluctuation regardless of the data collection interval. This suggests that the exposer's memory consumption is not heavily impacted by the frequency of data collection, maintaining a stable footprint across different intervals. In contrast, the broker shows a noticeable increase in memory usage as the interval approaches 1 second; however, it still consumes only 8.51 MB of memory, which is relatively small for such high-frequency data collection, this is due to the purge mechanism that we presented in the previous section. We were also interested in understanding the impact of the exposer and the message broker on overall computing resources in the computing node. To analyze this, we measured the total amount of computing resources consumed while varying the data collection time intervals in both K3S and KubeEdge container management platforms.

Figure 6 7 (left) shows the maximum CPU usage percentage for different time intervals. As observed, the CPU consumption increases as the collection interval shortens, with K3S consistently using more CPU than KubeEdge across all intervals, particularly at the 1-second interval where it reaches the highest value. However, even with the 1-second collection interval, CPU usage remains below 15% in the worst case, which is an acceptable level. Figure 6 (right) illustrates the percentage of memory used. Similarly, memory usage rises as the collection interval decreases. K3S shows higher memory usage compared to KubeEdge, especially at the 1-second interval, where memory usage peaks. Despite this, memory consumption remains within acceptable limits, with KubeEdge using less than 30% and K3S staying below 50%. This indicates that both platforms are capable of handling high-frequency data collection without significantly overloading the system's computing resources.

Lastly, we aimed to evaluate the response time of the RE. To achieve this, we compared the response time of the RE with to that of Prometheus [8], an open-source system monitoring and alerting toolkit, under different loads of simultaneous requests. The experiment was conducted over 50 trials, and the average results are presented in Figure 8. The latter shows that the response time for both the exposer API and the Prometheus API increases as the number of concurrent requests grows. Additionally, the results indicate that the Prometheus API takes approximately twice as long to respond compared to the exposer API. This demonstrates the efficiency of temporarily caching metric results in the broker which allows our exposer to fetch them more rapidly compared to solutions that relies

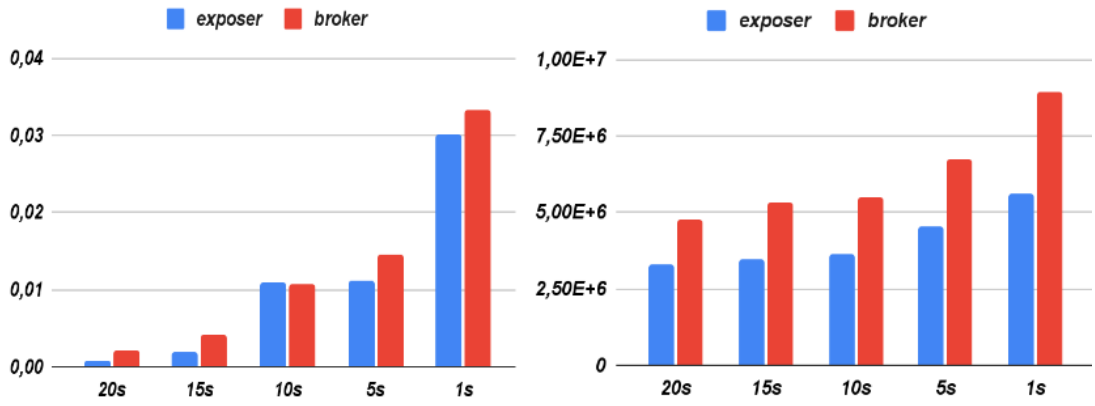

Fig. 6. CPU consumption (left) and memory usage (right) of the exposer and broker across varying data collection intervals.

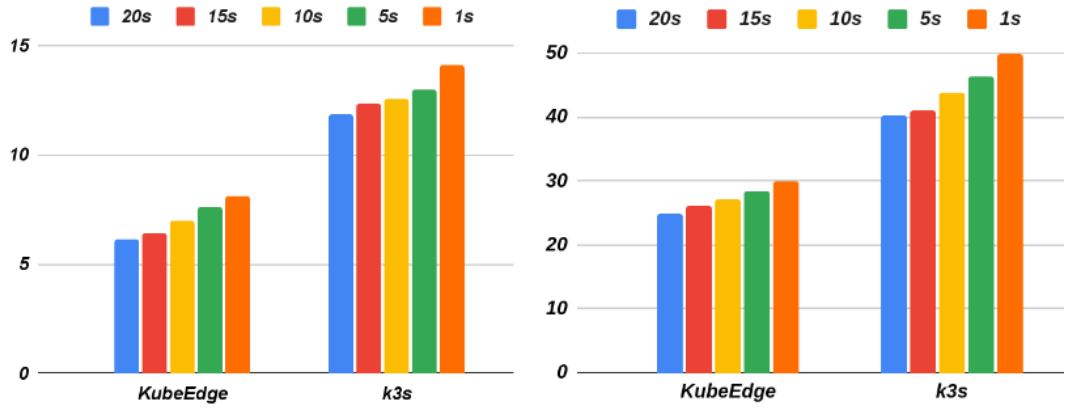

Fig. 7. Impact of the RE on overall resource footprint for KubeEdge and k3s: CPU usage (left) and memory usage (right).

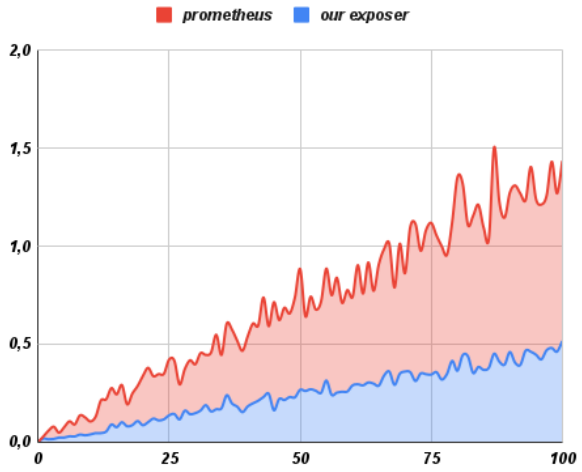

Fig. 8. Comparison of response times (in seconds) for the exposer API and Prometheus API under varying loads of simultaneous requests, demonstrating the efficiency of the exposer API and the impact of caching.

on a complete time series database.

V. Conclusion

In this paper, we introduced a novel resource exposure framework that enables a novel approach to orchestrate and manage the CEC by separating service orchestration from resource orchestration. The objective is to address the unique challenges posed by the heterogeneous and dynamic nature of edge and far-edge computing nodes. The proposed framework efficiently handles the volatility of resources in CEC using a self registration mechanism. Additionally, the Central Resource Discovery Module aggregates data from multiple REs, providing the RO with a global view of available resources and facilitating resource management across the continuum. Our performance evaluation demonstrated that the RE operates with minimal CPU and memory consumption, even in high-frequency data collection scenarios, making it highly suitable for low-resource environments such as edge nodes. Furthermore, the system's ability to maintain low-latency responses under multiple simultaneous requests highlights its potential for real-time applications.

In future work, we will leverage the exposure framework by integrating it with advanced decision-making algorithms for workload placement and resource allocation.

ACKNOWLEDGMENT

This work was partially supported by the European Union's Horizon Research and Innovation program under AC^3 project and grant agreement No 101093129.

REFERENCES

[1] P. A. Frangoudis, L. Yala, A. Ksentini, and T. Taleb, "An architecture for on-demand service deployment over a telco cdn," in *2016 IEEE International Conference on Communications (ICC)*, 2016, pp. 1–6.

[2] S. Arora, A. Ksentini, and C. Bonnet, "Cloud native lightweight slice orchestration (cliso) framework," *Comput. Commun.*, vol. 213, pp. 1–12, 2024.

[3] S.-H. Shen and A. Akella, "Decor: A distributed coordinated resource monitoring system," in *2012 IEEE 20th International Workshop on Quality of Service*, 2012, pp. 1–9.

[4] T. Nguyen and M. Pandey, "Dprof-distributed system profiling and tracing."

[5] M. Mekki, S. Arora, and A. Ksentini, "A scalable monitoring framework for network slicing in 5g and beyond mobile networks," *IEEE Transactions on Network and Service Management*, vol. 19, no. 1, pp. 413–423, 2022.

[6] J. R.-G. R. S. Sabine Randriamasy, Luis M. Contreras. (2024-07-07) Operational compute metrics. [Online]. Available: https://datatracker.ietf.org/doc/draft-rcr-opsawg-operational-compute-metrics/

[7] N. Toumi, M. Bagaa, and A. Ksentini, "Machine learning for service migration: A survey," *IEEE Commun. Surv. Tutorials*, vol. 25, no. 3, pp. 1991–2020, 2023.

[8] Prometheus. (n.d.) Prometheus: Monitoring system and time series database. [Online]. Available: https://prometheus.io/